UNIVERSIDAD DE BURGOS

Área de Tecnología Electrónica

Programación paralela e híbrida.

José María Cámara Nebreda, César Represa Pérez, Pedro Luis Sánchez Ortega
Programación Paralela e Híbrida. 2016
Área de Tecnología Electrónica
Departamento de Ingeniería Electromecánica
Universidad de Burgos

Introducción

En este curso vamos a dar un paso más en el desarrollo de aplicaciones paralelas. El objetivo sigue siendo explorar las posibildades existentes bajo el enfoque de la extracción del máximo rendimiento posible del hardware disponible.

En las asignaturas de Grado hemos visto cómo realizar una programación de datos paralelos en CUDA que extraiga el máximo rendimiento de un hardware específico como es la GPU. También hemos visto cómo realizar programación de paso de mensajes en MPI, lo cual nos permite extender la concurrencia de nuestras aplicaciones a entornos multicomputador.

Las arquitecturas actuales contemplan entornos multicomputador integrados por un cierto número de subsistemas multi-núcleo. En este escenario, la programación paralela permite la incorporación de otras formas de trabajo como es la comunicación mediante variables compartidas. Esto nos va a llevar a considerar en este curso el estándar OpenMP como alternativa o complemento a MPI.

Otro aspecto en el que vamos a profundizar es el relativo a la planificación. Entendiendo como tal, la influencia de las decisiones tomadas por el Administrador del sistema en cuanto a la gestión de las colas de procesos, en el rendimiento global de la máquina.

Como banco de pruebas nos vamos a centrar este curso en la resolución de un problema clásico, pero muy interesante, como es la generación de números primos. Emplearemos para ello una técnica sencilla y eficiente como es la criba de Eratóstenes.

Práctica 1: Implementación secuencial de la criba de Eratostenes.

OBJETIVOS

- Generar una aplicación secuencial de partida que nos permita evolucionar a versiones paralelas e híbridas posteriores.
- Descubrir y reolver las principales dificultades que plantea el procedimiento, especialmente en lo que afecta a su demanda de memoria.

CONCEPTOS TEÓRICOS

La criba de Eratóstenes se basa en la generación de una lista de números primos, hasta un máximo determinado, siguiendo un mecanismo de eliminación. Para ello se inicializan todos los números de la lista (excepto el 0 y el 1) como primos. Siguiendo una estructura de dos bucles anidados, se van eliminando (mancando como compuestos) todos los múltiplos de cada número primo hasta llegar al máximo configurado.

PRIMERA REALIZACIÓN PRÁCTICA

El primer elemento entragable de esta práctica es esta versión preliminar y secuecial de la criba. Se puede recurrir a este enlace para comprobar que los resultados son correctos. A partir de ella nos debemos plantear y responder las siguientes:

CUESTIONES

- ¿Cuál es el número más alto que nos permite generar nuestro método?
- ¿Cuál es la demanda de memoria para almacenar la lista en este caso?
- ¿Tenemos tiempos de ejecución elevados para esta versión?

SEGUNDA REALIZACIÓN PRÁCTICA

El objetivo es poder generar números primos hasta valores más altos de los que nos permiten los formatos de variable empleados habitualmente. Concretamente debemos poder alcanzar valores por encima de INT_MAX, que en este caso se sitúa en 2^{31}. La plicación debe ser capaz de trabajar con ellos y, sobre todo, no agotar la memoria RAM de nuestro computador. Esto nos puede llevar más de una iteración ya que nos iremos dando cuenta de las restricciones existentes a medida que las afrontemos.

CUESTIONES

- ¿Qué modificaciones han sido necesarias para llegar al objetivo marcado?
- ¿Hasta dónde se podría llegar con esta nueva implementación?
- ¿Cuál es el tiempo de ejecución en este caso?

Práctica 2: Implementación paralela (MPI)

OBJETIVOS

- Explorar alternativas de paralelización de paso de mensajes.
- Generar una aplicación lo más óptima posible que permita la generación de números primos en paralelo.

CONCEPTOS TEÓRICOS

Rendimiento de un sistema de tamaño variable

A continuación se van a definir una serie de conceptos manejados de forma habitual en la literatura sobre procesamiento en paralelo que permiten evaluar diferentes características de los sistemas y prever su rendimiento. Estos conceptos son los siguientes y se aplican a situaciones donde el número de procesadores utilizados para la ejecución de un programa varía a lo largo del tiempo de ejecución:

- **Grado de paralelismo (DOP)***:* Número de procesadores utilizados para ejecutar un programa en un instante de tiempo determinado. Esta función, **DOP** = *P(t)*, que nos da el grado de paralelismo en cada momento durante el tiempo de ejecución de un programa, se llama **perfil de paralelismo** de ese programa. No tiene porqué corresponder con el número de procesadores disponibles en el sistema (*n*). En las siguientes definiciones supondremos que se dispone de más procesadores que los necesarios para alcanzar el grado de paralelismo máximo de las aplicaciones, **máx**{*P(t)*} = *m* < *n*.

- **Trabajo total realizado**: Si Δ es el rendimiento, velocidad o potencia de cómputo de un solo procesador, medida en MIPS o MFLOPS, y suponemos que todos los procesadores son iguales, es posible evaluar el trabajo total realizado entre los instantes de tiempo t_A y t_B a partir de área bajo el perfil de paralelismo como:

$$W = \Delta \cdot \int_{t_A}^{t_B} P(t) \cdot dt \,.$$

  Normalmente el perfil de paralelismo es una gráfica definida a tramos (figura 2.1), por lo que se puede expresar el trabajo total realizado como:

$$W = \Delta \cdot \sum_{i=1}^{m} i \cdot t_i \,.$$

  En esta expresión t_i es el intervalo de tiempo durante el cual el grado de paralelismo es *i*, siendo *m* el máximo grado de paralelismo durante todo el tiempo de ejecución del programa.

Según esto, se cumple que la suma de los diferentes intervalos de tiempo es igual al tiempo de ejecución del programa:

$$\sum_{i=1}^{m} t_i = t_B - t_A .$$

- **Paralelismo medio**: Es la media aritmética del grado de paralelismo a lo largo del tiempo ejecución. Se expresa como:

$$\overline{P} = \frac{1}{t_B - t_A} \int_{t_A}^{t_B} P(t) \cdot dt = \frac{\sum_{i=1}^{m} i \cdot t_i}{\sum_{i=1}^{m} t_i} .$$

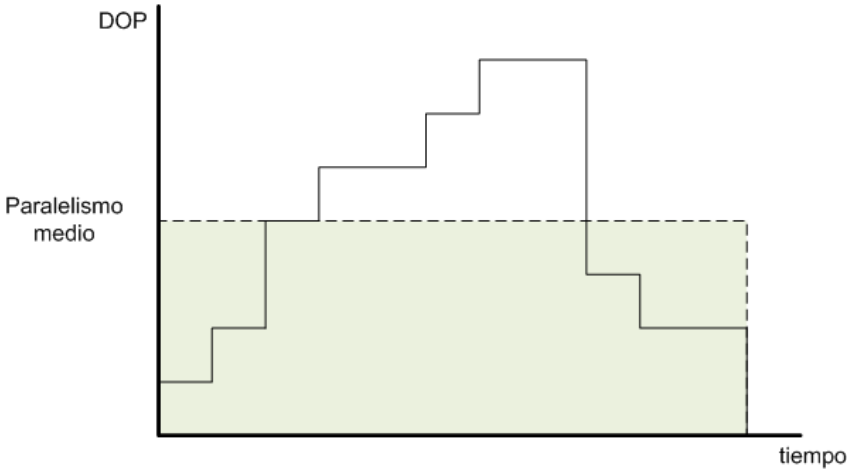


Figura 2.1: Perfil de paralelismo: representación gráfica de P(t).

- **Paralelismo disponible**: Máximo grado de paralelismo extraíble de un programa o aplicación, independientemente de las restricciones del hardware.

- **Máximo incremento de rendimiento alcanzable**: Sea $W_i = i \cdot \Delta \cdot t_i$ el trabajo realizado cuando **DOP** = *i*, de manera que $W = \sum_{i=1}^{m} W_i$.

En estas condiciones, el tiempo empleado por un solo procesador para realizar un trabajo W_i es $t_i(1) = \frac{W_i}{\Delta}$; para *k* procesadores es $t_i(k) = \frac{W_i}{k \cdot \Delta}$, y para infinitos procesadores es $t_i(\infty) = \frac{W_i}{i \cdot \Delta}$ (sólo trabajan "*i*" procesadores ya que el grado de paralelismo es *i*).

A partir de aquí se puede definir el **tiempo de respuesta** como:

$$T(1) = \sum_{i=1}^{m} t_i(1) = \sum_{i=1}^{m} \frac{W_i}{\Delta}$$

$$T(\infty) = \sum_{i=1}^{m} t_i(\infty) = \sum_{i=1}^{m} \frac{W_i}{i \cdot \Delta}.$$

El máximo rendimiento que puede proporcionar un sistema paralelo se da cuando en número de procesadores disponibles es ilimitado, por lo que el máximo incremento de rendimiento alcanzable (también denominado *speed-up asintótico*) se obtendrá del cociente entre éste y el rendimiento que proporciona un solo procesador. En términos de tiempos de respuesta se expresa como:

$$S_\infty = \frac{T(1)}{T(\infty)}.$$

Utilizando la definición de $W_i = i \cdot \Delta \cdot t_i$, la expresión del *speed-up asintótico* se puede poner como:

$$S_\infty = \frac{T(1)}{T(\infty)} = \frac{\sum_{i=1}^{m} \frac{W_i}{\Delta}}{\sum_{i=1}^{m} \frac{W_i}{i \cdot \Delta}} = \frac{\sum_{i=1}^{m} \frac{i \cdot \Delta \cdot t_i}{\Delta}}{\sum_{i=1}^{m} \frac{i \cdot \Delta \cdot t_i}{i \cdot \Delta}} = \frac{\sum_{i=1}^{m} i \cdot t_i}{\sum_{i=1}^{m} t_i} = \overline{P}.$$

Esto se puede simplificar diciendo que el máximo incremento de rendimiento alcanzable por un sistema paralelo que disponga de un número ilimitado de procesadores equivale al paralelismo medio intrínseco de la aplicación que se va a paralelizar. Lógicamente lo complicado es averiguar ese paralelismo intrínseco y lograrlo mediante la programación.

Rendimiento de un sistema con carga de trabajo fija

Existen otras formas de evaluar la ganancia en rendimiento de un sistema. Una de ellas es considerar situaciones donde una carga de trabajo fija se puede repartir entre un diferente número de procesadores. Por ejemplo, decimos que un trabajo *W*, ya sea un programa o un conjunto de ellos, se va a ejecutar en "modo *i*" si para ello se van a emplear *i* procesadores, siendo R_i el rendimiento o velocidad colectiva de todos ellos medida en MIPS o MFLOPS y $T(i) = W/R_i$ el tiempo de ejecución. Así, R_1 sería la velocidad de uno solo y $T(1) = W/R_1$ su correspondiente tiempo de ejecución. Supongamos que el trabajo *W* se realiza en *n* modos diferentes con una carga de trabajo diferente para cada modo, lo cual se refleja en un peso relativo f_i que se le asigna a cada uno:

$$W = W_1 + W_2 + \cdots + W_n = f_1 \cdot W + f_2 \cdot W + \cdots + f_n \cdot W$$

En estas condiciones podemos definir la ganancia en rendimiento como el cociente entre el tiempo $T(1)$ empleado por un único procesador (modo 1) y el tiempo total *T* empleado por los *n* modos (modo 1, modo2,…, modo *n*):

$$S = \frac{T(1)}{T}.$$

El tiempo total de ejecución se calcula a partir del rendimiento y de la carga de trabajo de cada uno de los modos de ejecución:

$$T = \sum_{i=1}^{n} \frac{f_i \cdot W}{R_i} = W \cdot \sum_{i=1}^{n} \frac{f_i}{R_i}.$$

De esta manera, la ganancia en rendimiento se puede poner como:

$$S = \frac{T(1)}{T} = \frac{1/R_1}{\sum_{i=1}^{n} f_i / R_i}.$$

En una situación ideal en la que no existen retardos por comunicaciones o escasez de recursos, considerando un trabajo unidad (W = 1) tendremos que si R_1 = 1, R_i = i :

$$S = \frac{1}{\sum_{i=1}^{n} f_i / i}.$$

En este contexto se enuncia la ley de Amdahl, donde suponemos que el trabajo se realiza en dos modos con pesos relativos α y $1-\alpha$. Una parte se realiza utilizando un procesador tal que $f_1 = \alpha$ y otra en "modo n" con n procesadores tal que $f_n = 1-\alpha$, lo que quiere decir que una parte del trabajo se va a realizar en modo secuencial y el resto empleando la potencia máxima del sistema. En estas condiciones:

$$S = \frac{1}{\frac{\alpha}{1} + \frac{1-\alpha}{n}} = \frac{n}{1+(n-1)\cdot\alpha}.$$

Esto implica que si α = 0, es decir, si idealmente todo el trabajo se puede realizar en "modo n" utilizando la potencia máxima del sistema, entonces:

$$S = n.$$

Sin embargo, si esto no es posible, el máximo incremento de rendimiento alcanzable (*speed-up asintótico*) cuando $n \to \infty$ es:

$$S_\infty = \lim_{n\to\infty} S = \frac{1}{\alpha}.$$

Dicho de otro modo, el rendimiento del sistema se encuentra limitado por el peso de la parte secuencial del trabajo.

Existen algunos parámetros que son de utilidad para evaluar las diferentes características de un sistema paralelo con n procesadores y son los siguientes:

- **Ganancia (*Speed-up*)**: Determina el grado de mejora del sistema:

$$S = \frac{T(1)}{T(n)} \leq n.$$

Se usa para indicar el grado de ganancia de velocidad de una computación paralela.

- **Eficiencia**: Determina el grado de aprovechamiento del sistema:

$$E = \frac{S}{n} = \frac{T(1)}{n \cdot T(n)} \leq 1.$$

La eficiencia mide la porción útil del trabajo total realizado por *n* procesadores. A partir de la eficiencia se puede definir la **escalabilidad**. Así, se dice que un sistema es escalable si la eficiencia *E*(*n*) del sistema se mantiene constante y cercana a la unidad.

- **Redundancia**: Es la relación entre el número total de operaciones que realiza el sistema completo con *n* procesadores y las que realizaría un solo procesador para realizar el mismo trabajo:

$$R = \frac{O(n)}{O(1)} \geq 1.$$

La redundancia mide el grado del incremento de la carga.

- **Utilización**: Refleja el grado de utilización del sistema completo:

$$U = R \cdot E \leq 1.$$

La utilización indica el porcentaje de recursos (procesadores, memoria, recursos, etc.) que se utilizan durante la ejecución de un programa paralelo

- **Calidad del sistema**:

$$Q = \frac{S \cdot E}{R}.$$

La calidad combina el efecto del speed-up, eficiencia y redundancia en una única expresión para indicar el mérito relativo de un cálculo paralelo sobre un sistema.

REALIZACIÓN PRÁCTICA

La paralelización de la criba plantea dos alternativas:

- Paralelizar el bucle exterior, es decir, el recorrido por los distintos números primos.
- Paralelizar el bucle interior, esto es, repartir la generación de múltiplos de cada uno.

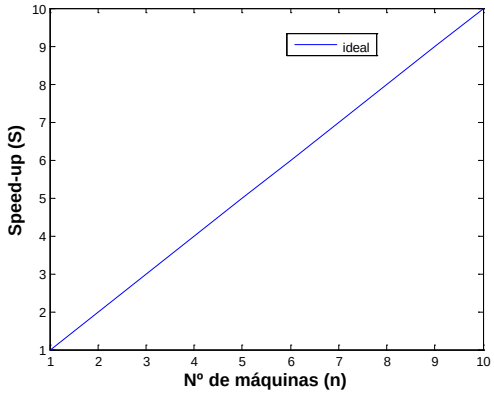
En esta práctica vamos a centrarnos en la primera opción. Una vez verificado que la generación de números primos en paralelo sigue siendo correcta se va a realizar una primera aproximación a la evaluación del rendimiento de la misma. Por el momento vamos a trabajar a nivel local, por lo que variaremos el número de procesos lanzados de 1 a 4. Generaremos números primos hasta el valor 10^9.

CUESTIONES

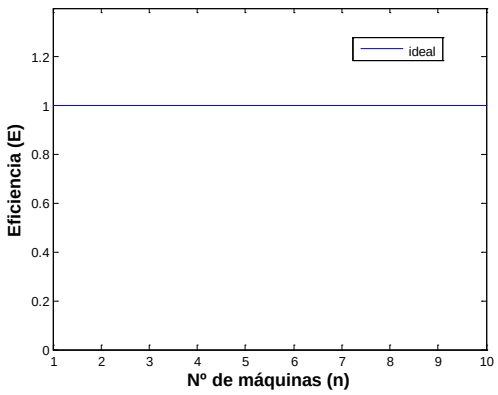
- ¿Está muy lejos el rendimiento obtenido del máximo teórico que se puede alcanzar?
- Evaluar el grado de acercamiento de la aplicación desarrollada a los máximos posibles, calculando para ello de forma aproximada la eficiencia, redundancia, utilización y la calidad del sistema.
- Estimar las causas de la desviación observada.
- Describir qué aspectos se deberían optimizar para obtener un mayor rendimiento.

Gráficas de rendimiento

Representar gráficamente la evolución del speed-up (*S*) y de la eficiencia (*E*) frente al número de máquinas y compararla con la evolución ideal (figura a y figura b, respectivamente).



(a)



(b)

Práctica 3: Implementación paralela (MPI) II

OBJETIVOS

- Explorar alternativas de paralelización de paso de mensajes.
- Generar una aplicación lo más óptima posible que permita la generación de números primos en paralelo.

REALIZACIÓN PRÁCTICA

La paralelización de la criba plantea dos alternativas:

- Paralelizar el bucle exterior, es decir, el recorrido por los distintos números primos.
- Paralelizar el bucle interior, esto es, repartir la generación de múltiplos de cada uno.

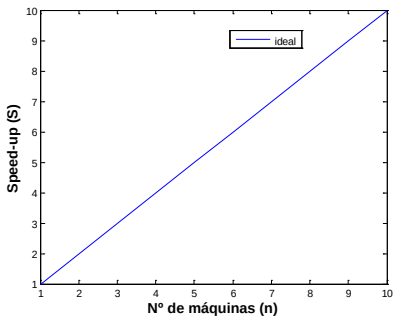
En esta práctica vamos a trabajar la segunda opción. Una vez verificado que la generación de números primos en paralelo sigue siendo correcta se va a realizar una primera aproximación a la evaluación del rendimiento de la misma. Por el momento vamos a trabajar a nivel local, por lo que variaremos el número de procesos lanzados de 1 a 4. Generaremos números primos hasta 10^9.

CUESTIONES

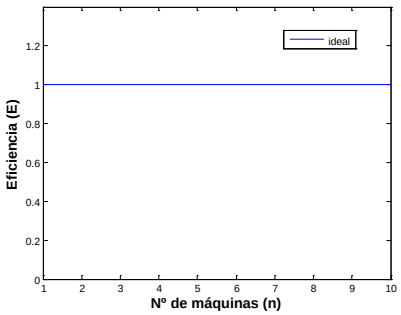
- ¿Está muy lejos el rendimiento obtenido del máximo teórico que se puede alcanzar?
- Evaluar el grado de acercamiento de la aplicación desarrollada a los máximos posibles, calculando para ello de forma aproximada la eficiencia, redundancia, utilización y la calidad del sistema.
- Estimar las causas de la desviación observada.
- Describir qué aspectos se deberían optimizar para obtener un mayor rendimiento.

Gráficas de rendimiento

Representar gráficamente la evolución del speed-up (*S*) y de la eficiencia (*E*) frente al número de máquinas y compararla con la evolución ideal (figura a y figura b, respectivamente).



(a)



(b)

Práctica 4: Introducción a la Programación de memoria compartida

OBJETIVOS

- Entender las limitaciones de la programación monohilo.
- Entender los beneficios de la programación multihilo.
- Conocer el funcionamiento básico de OpenMP.
- Aprender cómo incrustar hilos que comparten memoria dentro de procesos de memoria distribuida.

CONCEPTOS TEÓRICOS

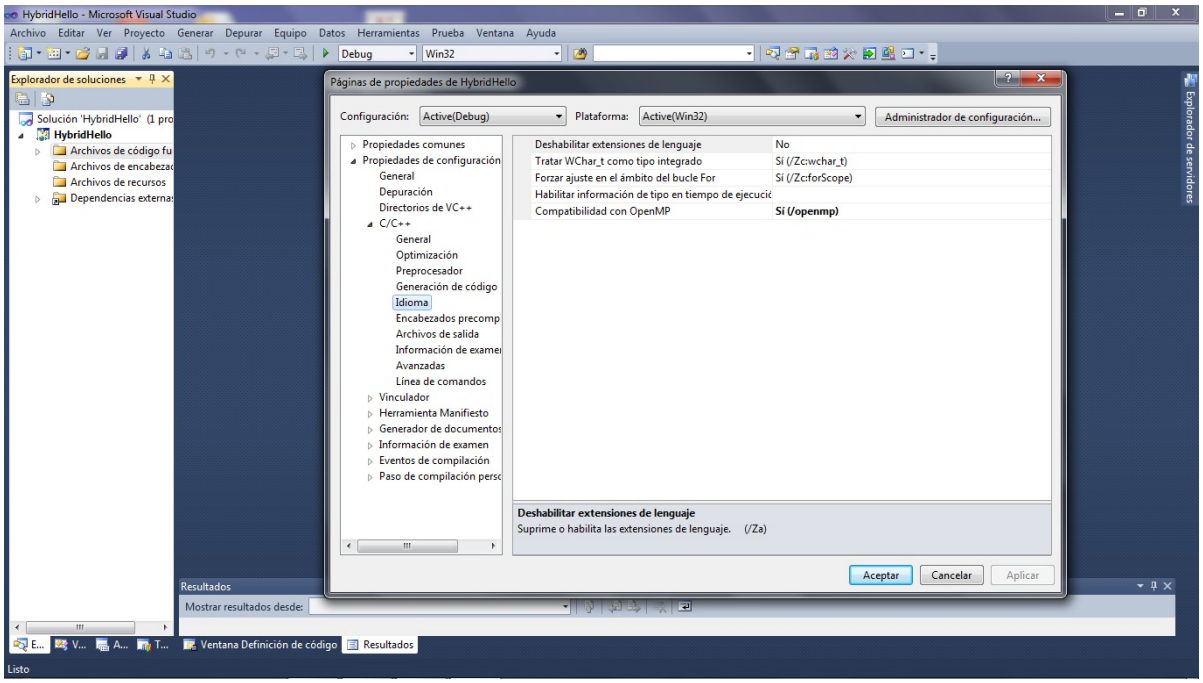
Limitaciones de la programación monohilo.

Hasta ahora hemos desarrollado programas de un solo hilo. Aunque hubiera varios hilos lanzados, cada proceso contenía solamente uno. Esto plantea notables limitaciones cuando el programa ha de correr en sistemas multinúcleo, los más habituales en la actualidad. Si lanzamos tantos procesos como procesadores haya disponibles, solamente uno de los núcleos del procesador va a estar ocupado. Se puede incrementar la eficiencia simplemente lanzando tantos procesos como núcleos. Incluso así, un análisis más profundo del uso de los recursos probablemente revelará que se mantiene un cierto grado de ineficiencia.

En los entornos de paso de mensajes como MPI, la información se intercambia mediante mensajes que se envían de un proceso fuente a otro destino. Estos mensajes contienen, aparte de la información en sí misma, algunos elementos de información adicionales: fuente, destino, etiqueta, cuenta, etc. Cuando los procesos fuente y destino se encuentran físicamente en procesadores distintos conectados a través de una red de área local, esto es lo correcto y prácticamente la única opción. Sin embargo, cuando los procesos se encuentran ubicados en el mismo procesador, no necesitan recurrir a un procedimiento tan complejo, ya que comparten una memoria común en la que ambos pueden leer y escribir. En este escenario, el uso de mensajes genera un considerable sobrecarga. Un sistema de memoria compartida ha de ser habilitado para optimizar el rendimiento.

Cómo incrustar hilos de memoria compartida en m procesos de paso de mensajes.

Hasta ahora hemos desarrollado programas de un solo hilo. Cada proceso MPI es monohilo. Podemos crear múltiples hilos de diferentes maneras pero vamos a seleccionar el entorno de programación de OpenMP por resultar una manera muy simple de llevarlo a cabo. Para habilitar la programación multihilo se deben realizar algunos ajustes en las propiedades del proyecto:



Así el proyecto está preparado para aceptar directivas de OpenMP, pero será necesario comentar algo sobre su funcionamiento para poder seguir avanzando.

REGIONES PARALELAS EN OpenMP

Los programs OpenMP son monohilo por defecto. Disponen de un hilomaestro que, en ciertas ocasiones se divide en varios hilos para llevar a cabo un conjunto de operaciones en paralelo. Estos trozos de código son denominados “regiones paralelas”.

```
#pragma omp parallel
{
	Parallel region
}
```

La función omp_get_thread_num() devuelve la identificación del hilo actual de 0 a N-1 pero, ¿cómo se establece el valor de N? Hay también varias opciones, pero aquí vamos a utilizar únicamente la definición estática:

```
#pragma omp parallel num_threads (N)
{
	Parallel region
}
```

Hasta aquí correcto, pero aún nos resta establecer los contenidos de la región paralela. Puede estar integrada por cualquier conjunto de instrucciones váildas, si bien en la mayoría de los casos se van a emplear para paralelizar bucles. Los bucles “for” son los candidatos ideaales:

```
#pragma omp parallel num_threads (N)
{
	#pragma omp for
		for(i=0;i<n;i++){
			Operations to be performed
		}
```

```
}
```

Las “n” operaciones a realizar se distribuirán entre los N hilos. Se espera con ello lograr una reducción del tiempo de ejecución en procesadores multinúcleo y/o multihilo.

Esto es un entorno de variable compartida pero, ¿dónde están las variables compartidas? Las variables declaradas fuera de la región paralela son compartidas. Las variables declaradas dentro son privadas de cada hilo. De todas formas, es posible hacer que una variable compartida se convierta en privada:

```
#pragma omp parallel num_threads (N) private (j)
{
      #pragma omp for
            for(i=0;i<n;i++){
                  Operations to be performed on variable j
            }
}
```

En este caso cada hilo va a tener su propia copia de “j” aunque haya sido declarada fuera de la región pero, ¿cuál será el valor de “j” en cada hilo? En el ejemplo anterior “j” no ha sido inicializada, independientemente del valor que pueda tener antes de entrar en la región. Si se desea empezar con el valor que tenía previamente a la región paralela en cada hilo, se debe modificar el código:

```
#pragma omp parallel num_threads (N) firstprivate (j)
{
      #pragma omp for
            for(i=0;i<n;i++){
                  Operations to be performed on variable j
            }
}
```

De la misma forma, podemos necesitar que el hilo maestro tenga constancia de los cambios que ha sufrido “j” dentro de la región. Podemos conseguir que, al finalizar la región “j” tome el último valor que adquirió dentro:

```
#pragma omp parallel num_threads (N) firstprivate (j) lastprivate (j)
{
      #pragma omp for
            for(i=0;i<n;i++){
                  Operations to be performed on variable j
            }
}
```

Para concluir esta breve introducción vamos a estudiar una capacidad adicional de OpenMP. No será difícil de entender ya que existe una equivalente en MPI que ya ha sido utilizada. Se trata de la operación de reducción. Se aplica al caso de que una variable compartida es modificada hacia valores diferentes en los distintos hilos de ejecución. A veces, el valor final de esta variable tiene que ser obtenido como combinación de los valores generados por cada uno de los hilos. Vamos a ver un ejemplo sencillo:

```
#pragma omp parallel num_threads (N)
{
        #pragma omp for reduction(+:sum)
              for (i=0;i<n;i++){
```

```
                    sum=sum+(a[i]);
                }
}
```

Resulta obvio que lo que se pretende es obtener el valor final de la variable “sum” que deberá ser el resultado de las “n” sumas realizadas sobre ella. La reducción va a tomar el último valor generado por cada hilo y va a calcular la suma de todos ellos. Para que esto sea posible, se genera una copia privada de la variable en cada uno.

REALIZACIÓN PRÁCTICA

Volviendo atrás, a la solución secuencial, vamos a modificarla para que el bucle exterior (el que va recorriendo los números primos) se ejecute de forma paralela.

1. Lanzaremos 4 hilos de ejecución (tantos como núcleos tiene nuestro procesador).
2. Comparar los tiempos de ejecución con los de la apliación secuencial y la apliación paralela MPI.
3. Justificar los resultados obtenidos prestando especial atención al concepto de “redundancia”.

CUESTIONES

- ¿Qué alternativa de las contempladas hasta ahora proporciona el rendimiento más alto?
- ¿Cuáles pueden ser las causas?
- ¿Son estos resultados los esperados? ¿Por qué?

Práctica 5: Programación de memoria compartida II

OBJETIVOS

- Entender las limitaciones de la programación monohilo.
- Entender los beneficios de la programación multihilo.
- Conocer el funcionamiento básico de OpenMP.
- Aprender cómo incrustar hilos que comparten memoria dentro de procesos de memoria distribuida.

REALIZACIÓN PRÁCTICA

Volviendo atrás, a la solución secuencial, vamos a modificarla para que el bucle interior (el que va generando múltiplos de cada número) se ejecute de forma paralela.

1. Lanzaremos 4 hilos de ejecución (tantos como núcleos tiene nuestro procesador).
2. Comparar los tiempos de ejecución con los de la apliación secuencial, la apliación paralela MPI y la aplicación OpenMP de la práctica anterior.
3. Justificar los resultados obtenidos prestando especial atención al concepto de “redundancia”.

CUESTIONES

- ¿Qué alternativa de las contempladas hasta ahora proporciona el rendimiento más alto?
- ¿Cuáles pueden ser las causas?
- ¿Son estos resultados los esperados? ¿Por qué?

Práctica 6: Programación de memoria compartida III

OBJETIVOS

- Abundar en el concepto de programación de memoria compartida explorando soluciones de planificación dinámica.

CONCEPTOS TEÓRICOS

Sincronización.

El procedimiento de sincronización por defecto introduce una barrera al final de las regiones paralelas de manera que la ejecución no continúa hasta que todos los hilos alcanzan ese punto. Esto es razonable pero en ciertos casos puede ser interesante evitar esta restricción. Se puede hacer así mediante la cláusula “nowait”:

```
#pragma omp parallel num_threads (N)
{
      #pragma omp for nowait
            for(i=0;i<n;i++){
                  Operations to be performed on variable j
            }
}
```

En este ejemplo concreto, no plantea ninguna diferencia práctica pero en caso de que hubiera un nuevo bucle paralelo a continuación permitiría que los hilos pudieran entrar en él con la mayor rapidez.

Planificación.

Hasta el momento se ha asumido que el trabajo a realizar se reparte entre los diferentes hilos participantes de manera equitativa. Esto es correcto pero, incluso siendo así puede haber diferentes posibilidades en el reparto que pueden tener consecuencias significativas en el rendimiento.

La política de planificación por defecto divide el número de iteraciones entre el número de hilos proporcionando a todos la misma cantidad de trabajo si es posible. Este reparto se realiza previamente a la ejecución; no se realizan cambios en tiempo de ejecución. OpenMP permite definir el tamaño de los bloques de trabajo (número de iteraciones en este caso); cada implementación de OpenMP decide cómo asignar estos bloques a cada hilo.

```
#pragma omp parallel num_threads (N)
{
      #pragma omp for schedule(static,10)

            for(i=0;i<n;i++){
                  Operations to be performed on variable j
```

```
            }
}
```

En este ejemplo se reparten bloques de 10 iteraciones. Los últimos bloques se hacen más pequeños si la división no es exacta.

Las políticas estáticas no permiten asignar trabajo dinámicamente a los hilos a medida que terminan el que ya tenían asignado. Esto supone una pérdida de eficiencia que debería ser evitada. Para ello es posible aplicar políticas dinámicas.

```
#pragma omp parallel num_threads (N)
{
      #pragma omp for schedule(dynamic,10)

            for(i=0;i<n;i++){
                  Operations to be performed on variable j
            }
}
```

En este ejemplo los hilos obtienen nuevos bloques tan pronto como finalizan el cálculo en curso.

REALIZACIÓN PRÁCTICA

Se va a continuar el trabajo realizado en las dos prácticas anteriores. Disponemos ya de resultados que proporciona la planificación estática, por lo que vamos a añadir resultados nuevos:

- Utilizar de nuevo la planificación estática pero especificando bloques de 10.
- Repetir el experimento utilizando bloques de 100.
- Cambiar ahora a planificación dinámica con bloques de 10.
- Utilizar de nuevo la planificación dinámica con bloques de 100.

Realizar las experiencias anteriores tanto para la paralelización del bucle exterior como del interior.

Comparar todos los resultados para determinar cuál es la política más adecuada y tratar de explicar por qué. Comparar los reaultados con los proporcionados por la paralelización de paso de mensajes.

REFERENCIAS:

OPENMP APPLICATION PROGRAM INTERFACE. Disponible en:
http://www.openmp.org/mp-documents/spec30.pdf

Práctica 7: Programación híbrida

OBJETIVOS

- En programación híbrida se puede recurrir a múltiples combinaciones de número de procesos e hilos. Vamos a intentar descubrir cuál es la mejor.
- El paso de mensajes y la memoria compartida implican diferentes modelos de programación y un uso distinto de los recursos hardware. Debemos saber entones cuál es la técnica más eficiente y adecuada.

CONCEPTOS TEÓRICOS

No se van a introducir nuevos conceptos teóricos en esta práctica.

REALIZACIÓN PRÁCTICA

- A la vista de los resultados de las prácticas anteriores se ha de determinar qué combinación de paralelización de memoria compartida / paso de mensajes puede ser la más óptima.
- Una vez determinada, se propone realizar una batería de pruebas de rendimiento sobre diferentes configuraciones hardware. Dado que parece lógico mantener el número de hilos a 4, coincidiendo con el número de núcleos por procesador, iremos incrementando el número de máquinas de 1 a 6 y en la misma medida el número de procesos.

Representar gráficamente la evolución del speed-up (*S*) y de la eficiencia (*E*) frente al número de máquinas y compararla con la evolución ideal. Comparar también los resultados obtenidos con los proporcionados por la programación de paso de mensajes.

> Al asignar recursos a los trabajos en mútiples máquinas hay que tener en cuenta que la asignación por núcleos distribuirá los procesos sin tener en cuenta en qué máquina está cada uno. Para que los procesos MPI que lanzamos en programación híbrida se distribuyan de forma homogénea será necesario cambiar la unidad de asignación por defecto a “node”.

REFERENCIAS:

Comparing the OpenMP, MPI, and Hybrid Programming Paradigms on an SMP Cluster
Gabriele Jost and Haoqiang Jin and Dieter An Mey and Ferhat F. Hatay
NAS Technical Report NAS-03-019, November 2003.

Performance analysis of pure MPI versus MPI+OpenMP for Jacobi Iteration and a 3D FFT on the Cray XT5. Olga Weiss. Iowa State University. 2012.

Práctica 8: Planificación de trabajos

OBJETIVOS

- Conocer las políticas de planificación disponibles.
- Analizar su repercusión en el rendimiento global del sistema.

CONCEPTOS TEÓRICOS

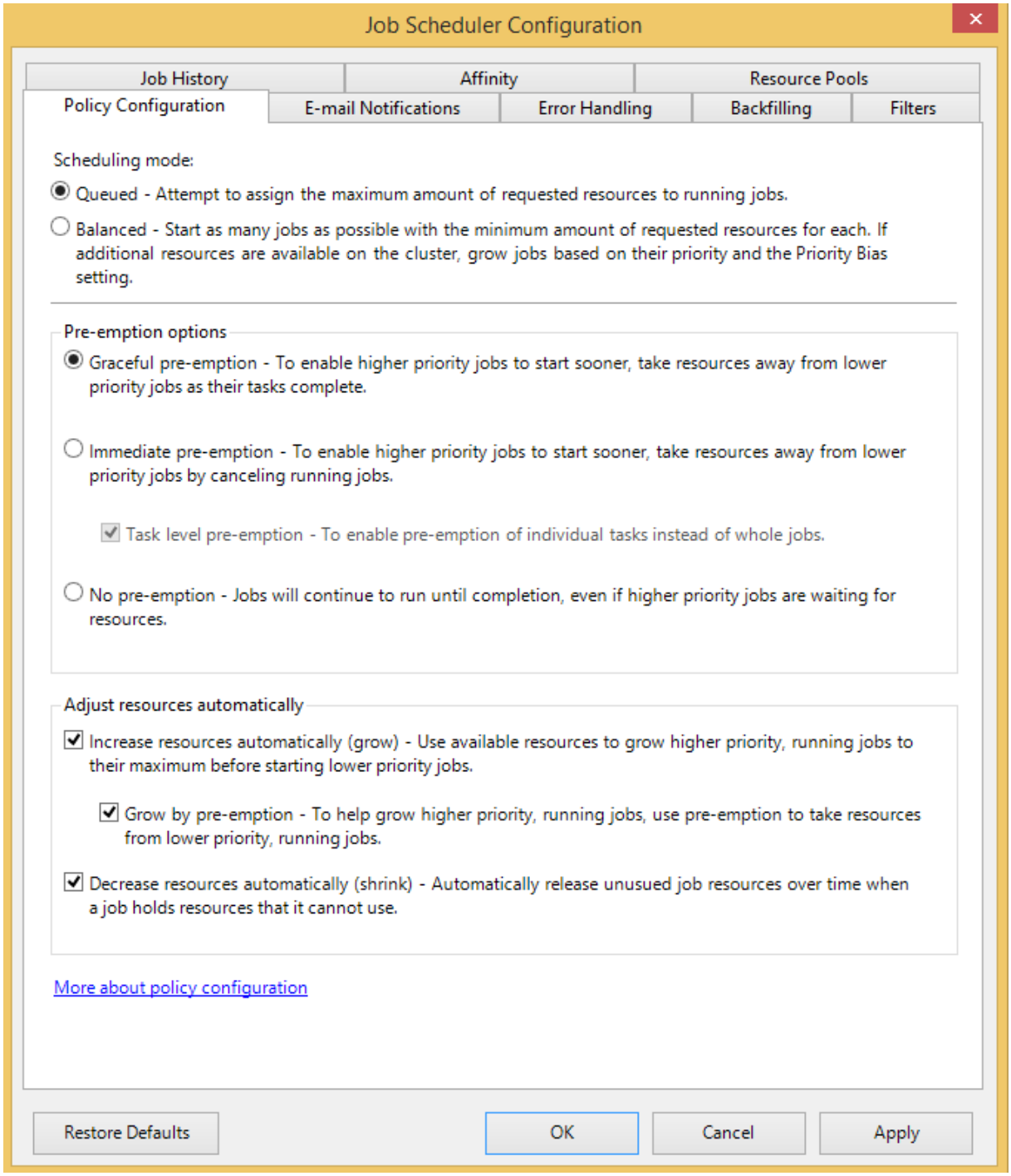
El administrador del Sistema se encarga de configurar políticas de planificación eficientes. Su objetivo es optimizer el rendimiento del Sistema, lo cual no es un concepto obvio. De hecho existen diferentes formas de interpretar el rendimiento, lo cual da lugar a objetivos diferentes también:

- Maximizar la utilización del sistema.
- Minimizar el tiempo de ejecución de los trabajos (wall time).
- Maximizar el throughput (trabajos realizados por unidad de tiempo).

Desde el punto de vista del usuario, el "wal time" es lo que suele se más importante pero el administrador no va a ocuparse de un único usuario priviliegiado, sino que tratará de obtener el máximo partido del conjunto del sistema.

El planificador de trabajos de Windows HPC proporciona diversas opciones de planificación. Lo primero que se ha de decider es si se quiere seguir una política encolada o balanceada en la planificación de los trabajos:

- Encolada (queued): el planificador va a intentar otorgar el máximo de los recursos solicitados por los trabajos y sus tareas. Cuando todos los recursos estén agotados, los trabajos siguientes tendrán que esperar en la cola. Como se muestra en la figura, varias sub-opciones acompañan a esta decisión.



Figua 7.1.Política de planificación balanceada.

Una de ellas es cómo manejar la precedencia. Si seleccionamos la opción elegante (graceful), los trabajos más prioritarios van a tomar recursos de otros, pero sólo cuando sus tareas vayan finalizando. La precedencia inmediata permite cancelar trabajos menos prioritarios en curso para dar servicio a otros más prioritarios. También es posible deshabilitar la precedencia. Se puede automatizar la Gestión de los recursos. Así, los trabajos más prioritarios recibirían más recursos, de acuerdo con la política de predecencia adoptada, hasta alcanzar el máximo solicitado, antes de que trabajos menos prioritarios puedan comenzar. Podrían incluso crecer tomando recursos de trabajos de menor prioridad que ya

estén en curso. Finalmente, se puede establecer que se retiren recursos a trabajos que ya no vayan a tener opción de utilizarlos.

- Balanceada (balanced): el planificador intentará arrancar tantos trabajos como sea posible. Para ello reducirá el número de recursos asignados a cada uno sin sobrepasar el mínimo solicitado por el usuario. De esta forma, siempre que haya recursos disponibles, se lanzarán nuevos trabajos en lugar de incrementar los recursos asignados a trabajos ya en curso. Como en el caso anterior, aparecen una serie de sub-opciones.

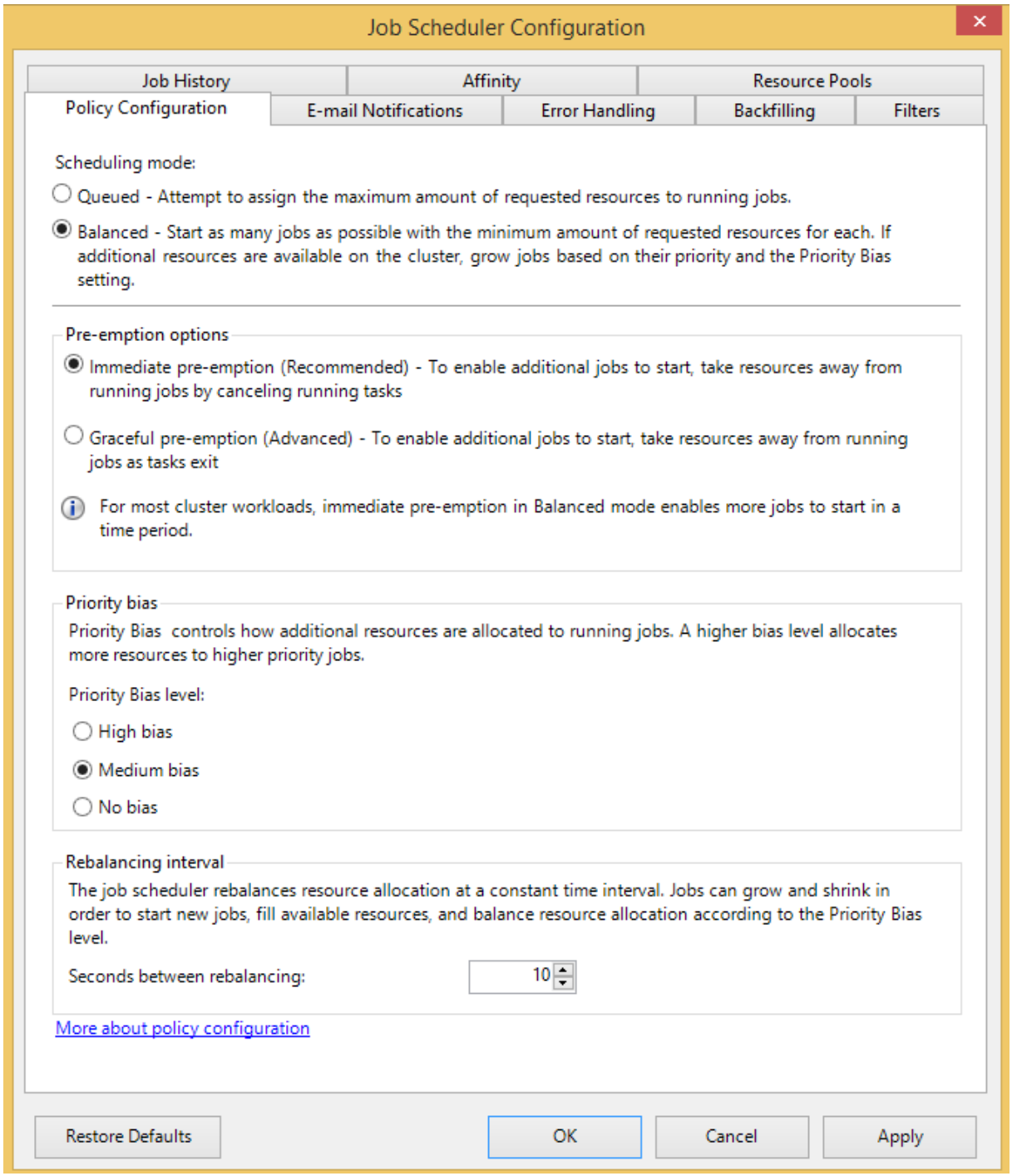


Figure 7.2. Balanced scheduling policy.

De Nuevo, la precedencia se puede seleccionar como elegante o inmediata, pero no se puede deshabilitar. Cuando se trata de asignar recursos adicionales a trabajos en curso, la decision de cómo hacerlo está condicionada por la prioridad de los mismos. La influencia de la prioridad en la decision es ajustable. También es possible ajustar el interval de rebalanceo de recursos.

El usuario asigna la prioridad a cada trabajo en el momento de su configuración. Dentro de un mismo trabajo todas las tareas comparten el mismo nivel de prioridad. En esta práctica vamos a trabajar con trabajos únicos, por lo que ignoraremos todo lo que tiene que ver con la prioridad por el momento.

REALIZACIÓN PRÁCTICA

Vamos a configurar un trabajo integrado por multiples tareas MPI. Cada tarea consistirá en la ejecución del cálculo de números primos hasta 10^9. El número de procesos se incrementará desde 4 hasta el número de núcleos disponibles en el Sistema, en incrementos de 4 en 4.

El trabajo se lanzará bajo una política encolada y bajo una política balanceada. En ambos casos se lanzará ordenando las tareas por número de procesos, de menor a mayor y de mayor a menor.

Para cada caso, se mostrará en una table los recursos asignados a cada tarea, su tiempo de ejecución y el tiempo de ejecución de todo el trabajo.

¿Qué politica de planificación resulta ser major para este tipo de carga? Intentar explicar por qué a partir de los contenidos de las tablas.

Repetir el proceso para la solución híbrida.

Práctica 9: Planificación de trabajos II

OBJETIVOS

- Comprender la diferencia entre tareas y trabajos.
- Trabajar con difierentes niveles de prioridad y distintas políticas de planificación.

CONCEPTOS TEÓRICOS

En esta práctica vamos a trabajar con niveles de prioridad distintos, por lo que aparecerán disponibles nuevas opciones de configuración. Los niveles de prioridad se pueden asignar a los trabajos, ya sea porque el usuario los considera de distinta importancia o para buscar una mayor eficiencia en su ejecución. En esta práctica la motivación va a ser la segunda. Los niveles de prioridad asignables a cada trabajo son:

- Muy alta (Highest).
- Alta (Above normal).
- Normal.
- Baja (Below normal).
- Muy baja (Lowest).

En este momento aparece una nueva cuestión: es posible que se exceda el máximo número de conexiones permitidas a la carpeta compartida (hasta 20). La solución se encuentra en el uso de carpetas compartidas en el servidor, cuyo sistema opertivo permite un número de conexiones prácticamente ilimitado. Para ello se debe ajustar la carpeta compartida proporcionada por el admisitrador como “directorio de trabajo” y copiar el ejecutable de la aplicación a dicha carpeta.

EJERCICIO PRÁCTICO

En primer lugar vamos a repetir los experimentos realizados en la práctica anterior pero lanzando multiples trabajos de una sola tarea en lugar de un solo trabajo con multiples tareas. Por el momento mantendremos el nivel de prioridad normal que adquieren por defecto. Reconstruir las tablas de resultados de la práctica anterior con los nuevos datos y comparar ambas.

¿Cómo son los tiempos de ejecución respecto al caso anterior? ¿Qué otras circunstancias se producen ahora? ¿Cómo se pueden superar?

Manteniendo los ajustes de planificación por defecto, modificar los niveles de prioridad de los trabajos a los que se considere más adecuados y repetir el procedmiento anterior.

¿Se ha conseguido mejorar el rendimiento? ¿Por qué puede ser?

Práctica 10: Planifiación de trabjos III

OBJETIVOS

- Comprender el concepto de precedencia.
- Comprobar la influencia de la precedencia en el rendimiento del sistema.

CONCEPTOS TEÓRICOS

La precedencia permite que trabajos más prioritarios interrumpan a otros menos. Como ya se ha comentado, esto se puede llevar a cabo de diferentes formas. Dado que el objetivo es el rendimiento del sistema, la prioridad se asignará de manera que se minimice el tiempo de ejecución global.

EJERCICIO PRÁCTICO

Agrupar las tareas lanzadas en las practices anteriores en 2 trabajos. En uno de ellos se situarán las tareas que demandan menos recursos y se le dará mayor prioridad. En el otro trabajo, con prioridad menor, se encontrarán el resto de las tareas.

De Nuevo bajo las políticas encolada y balanceada, repetir los experimentos habituales variando las políticas de precedencia.

Construir de Nuevo las tablas de resultados y compararlas.

Decidir qué política de precedencia es la más aconsejable para este tipo de carga de trabajo.

Comparar los resultados obtenidos mediante la política de planificación encolada con las opciones “Adjust resources automatically” activadas y sin activar.

Comparar los resultados obtenidos mediante la política de precedencia balanceada variando la influencia de la prioridad en la precedencia.

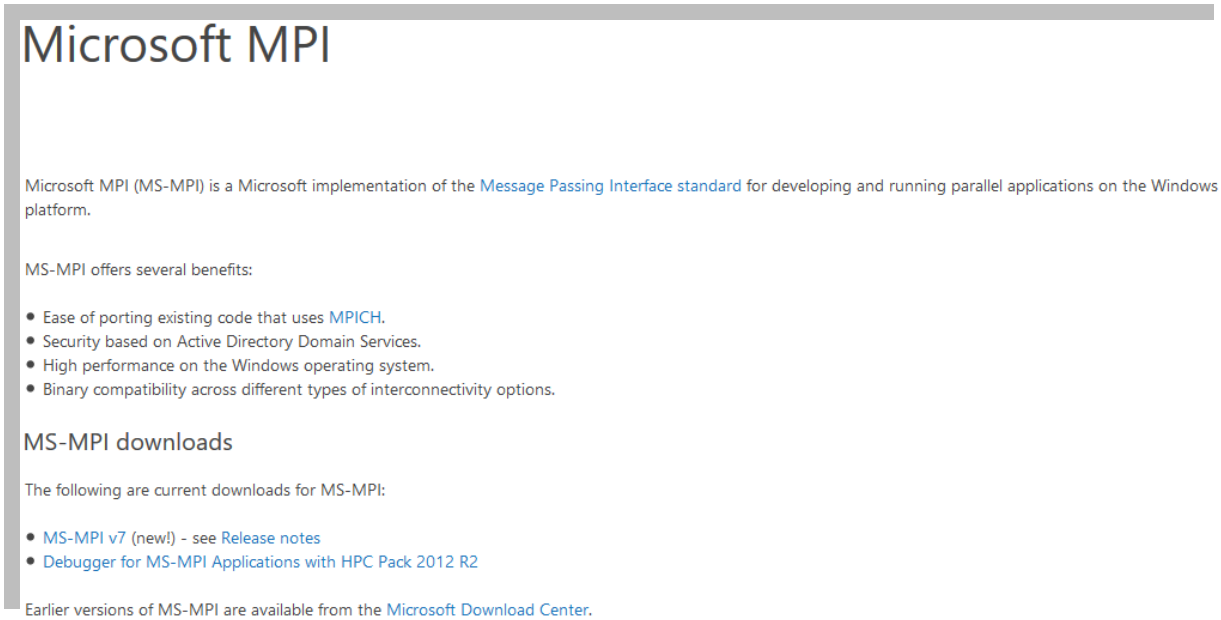
Apéndice A: Configuración de un Proyecto en Visual Studio.

En esta sección vamos preparar un proyecto que permita construir una aplicación MPI dentro del nuevo entorno de trabajo proporcionado por Microsoft Visual Studio.

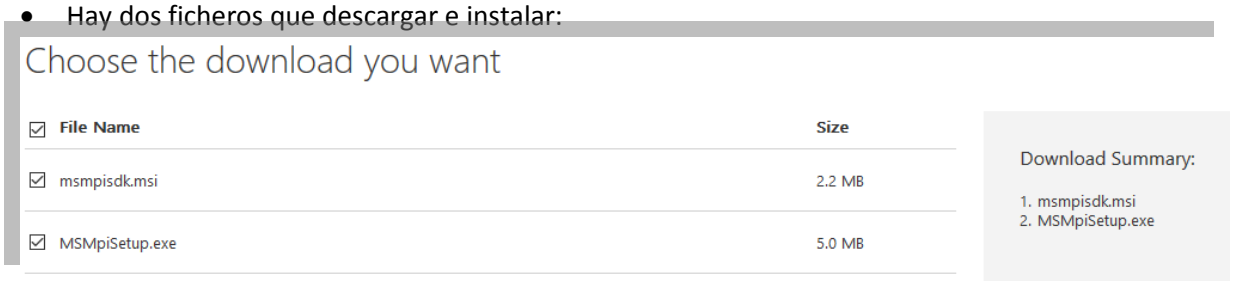
Configuración de MS-MPI.

Microsoft nos proporciona una implementación de MPI que podemos utilizar para llevar a cabo nuestras aplicaciones paralelas. Seguiremos los siguientes pasos para ponerla en funcionamiento:

- Descargar MS-MPI v de su localización web:



- Hay dos ficheros que descargar e instalar:

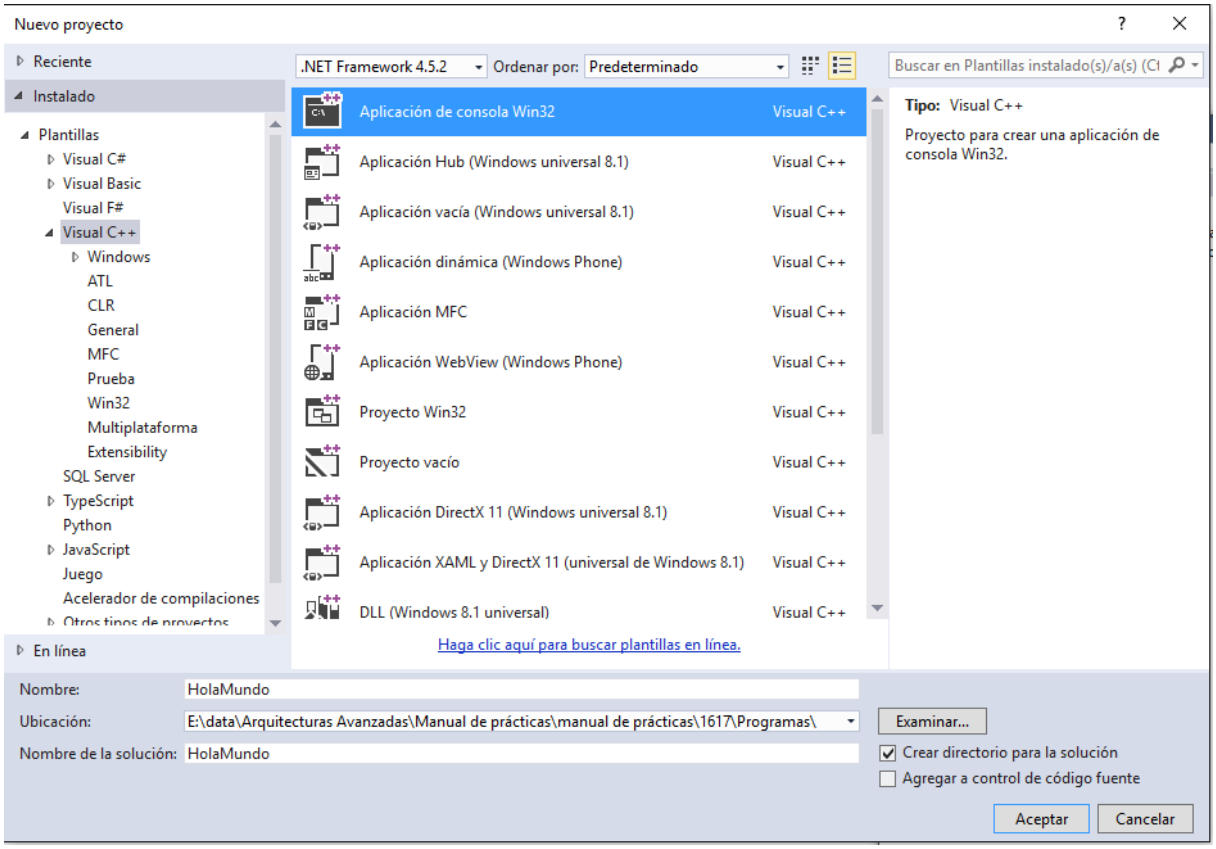


- Cada paquete crea una nueva carpeta: Archivos de Programa > Microsoft MPI y Archivos de Programa > Microsoft SDKs > MPI.

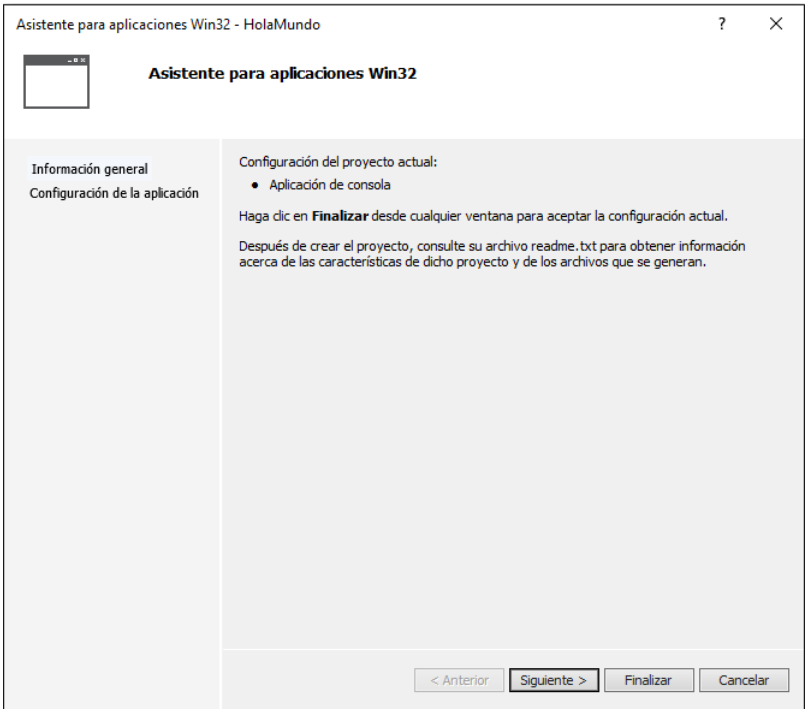
Configuración del proyecto.

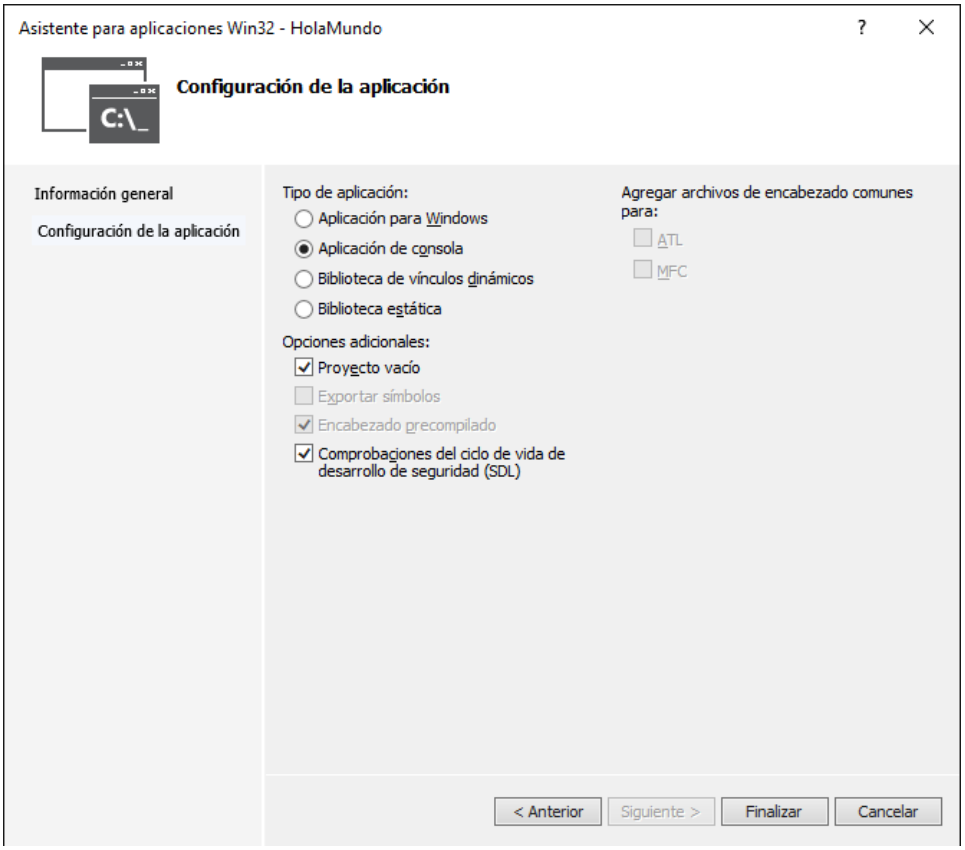
Ahora ya podemos crear una aplicación. Lo haremos a través de MS Visual Studio siguiendo los pasos que se detallan a continuación:

- Crear un **nuevo proyecto y solución**. Se les puede dar a ambos el mismo nombre:

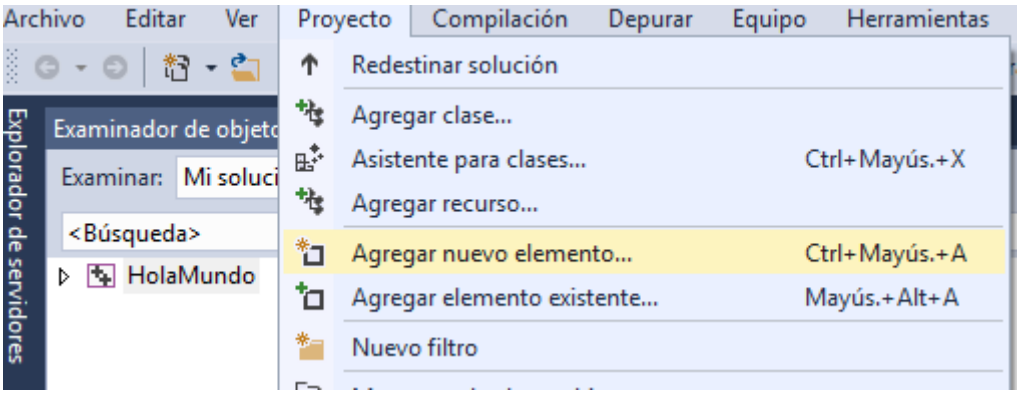


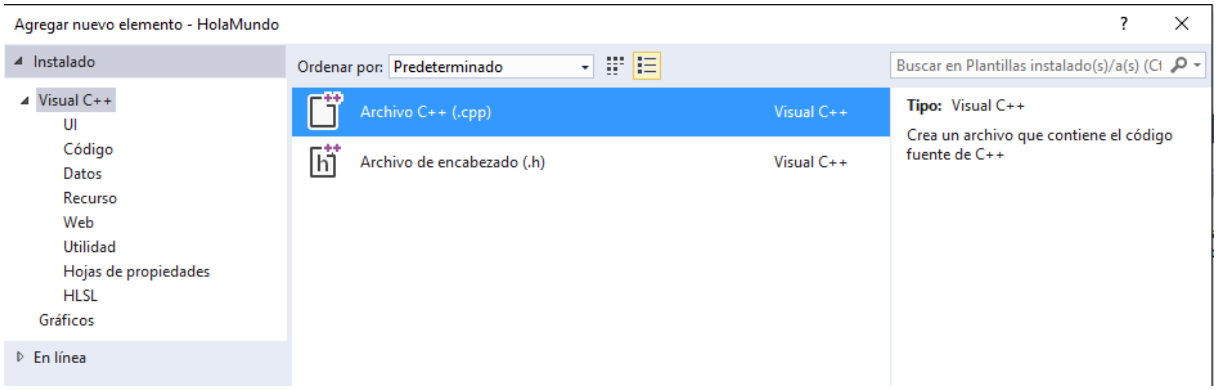
- Configurarlo como **proyecto vacío** (“*Empty project*”):



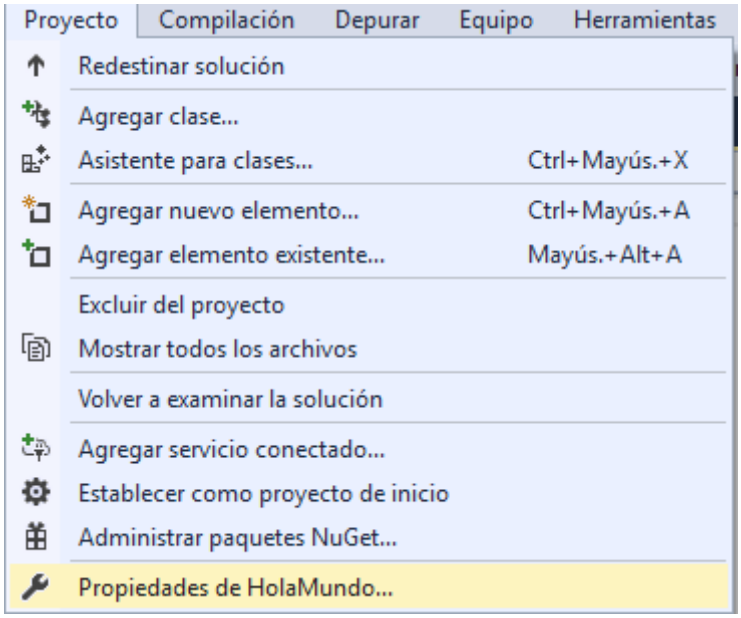


- Una vez creado el proyecto y la solución, añadir un fichero de código como **nuevo elemento** (“*Add New Item*”):

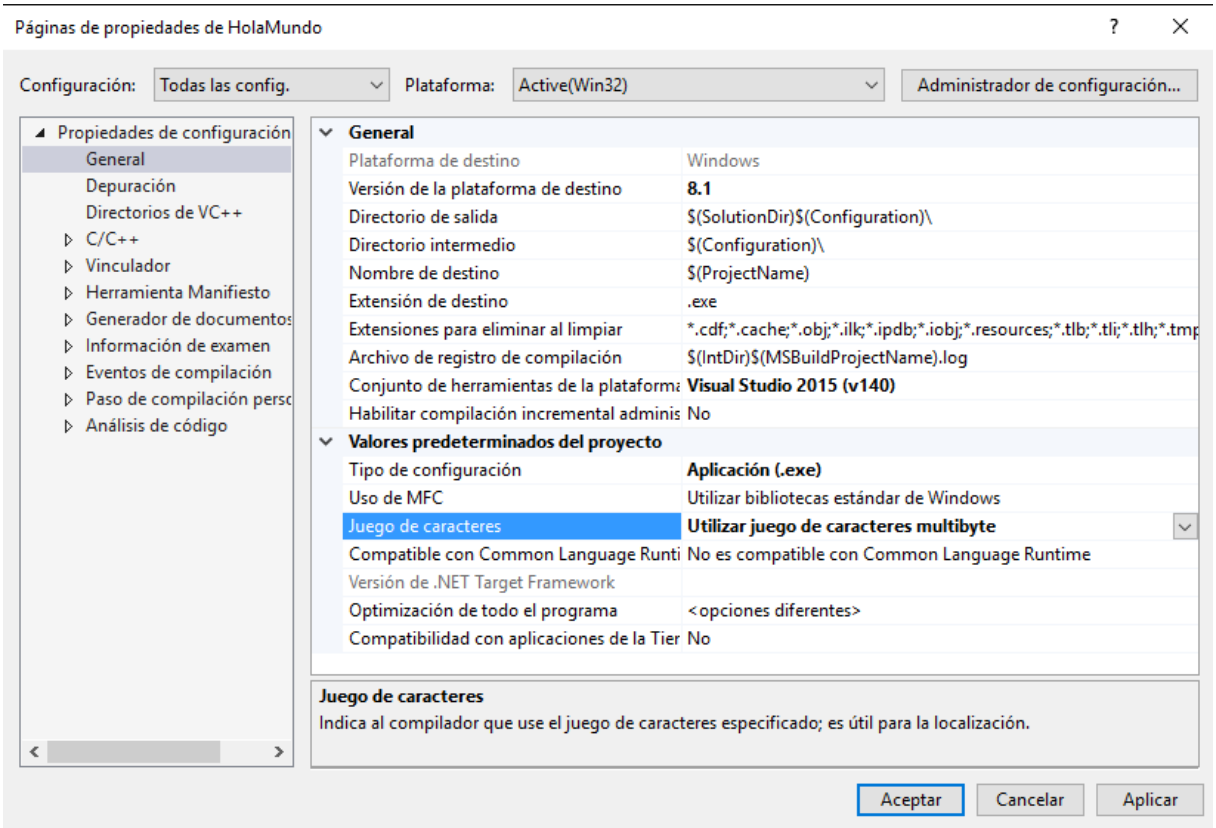




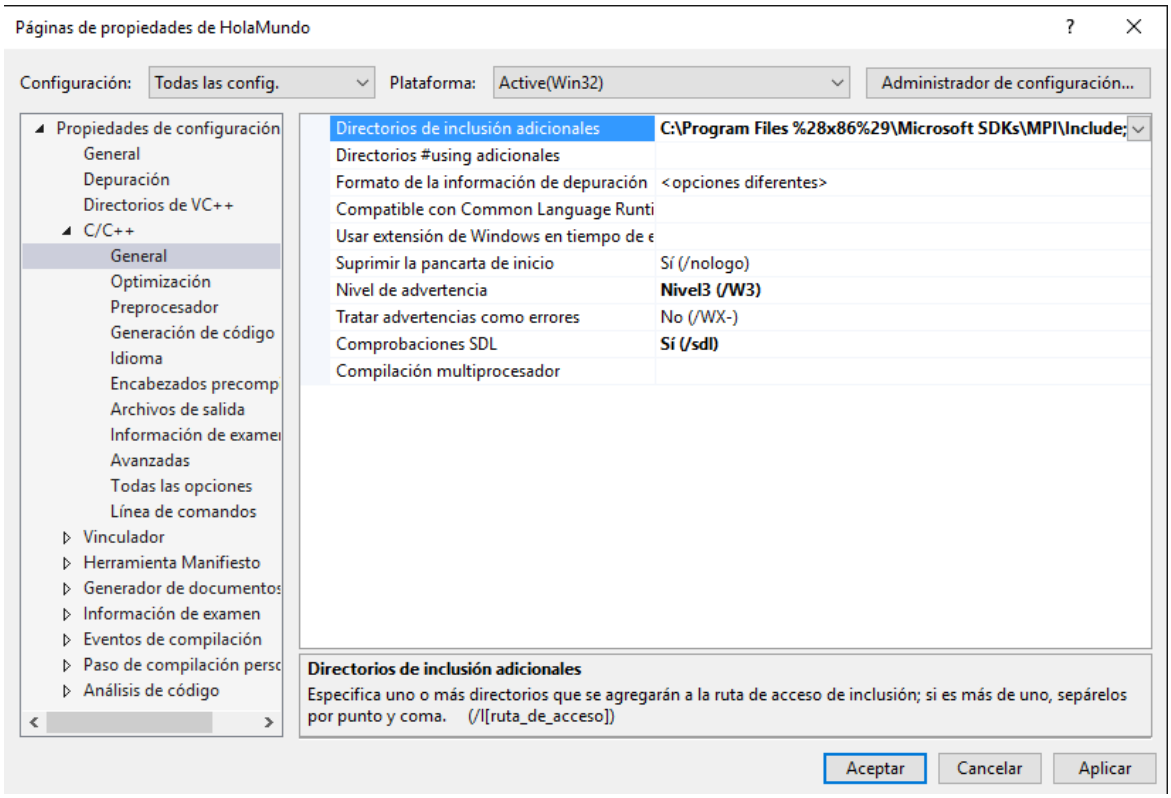
- A continuación, y no antes, se van a ajustar las **propiedades del proyecto** (“*Properties*”):



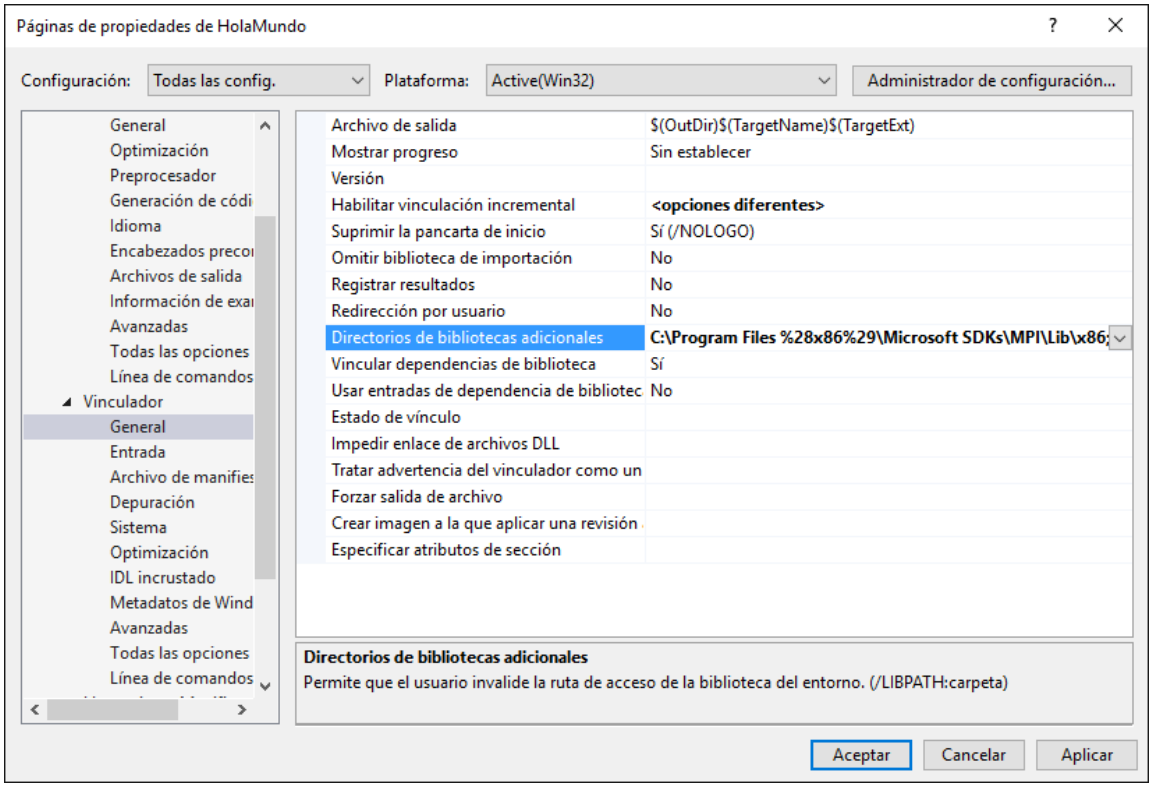
1. Configurar el juego de caracteres multibyte.



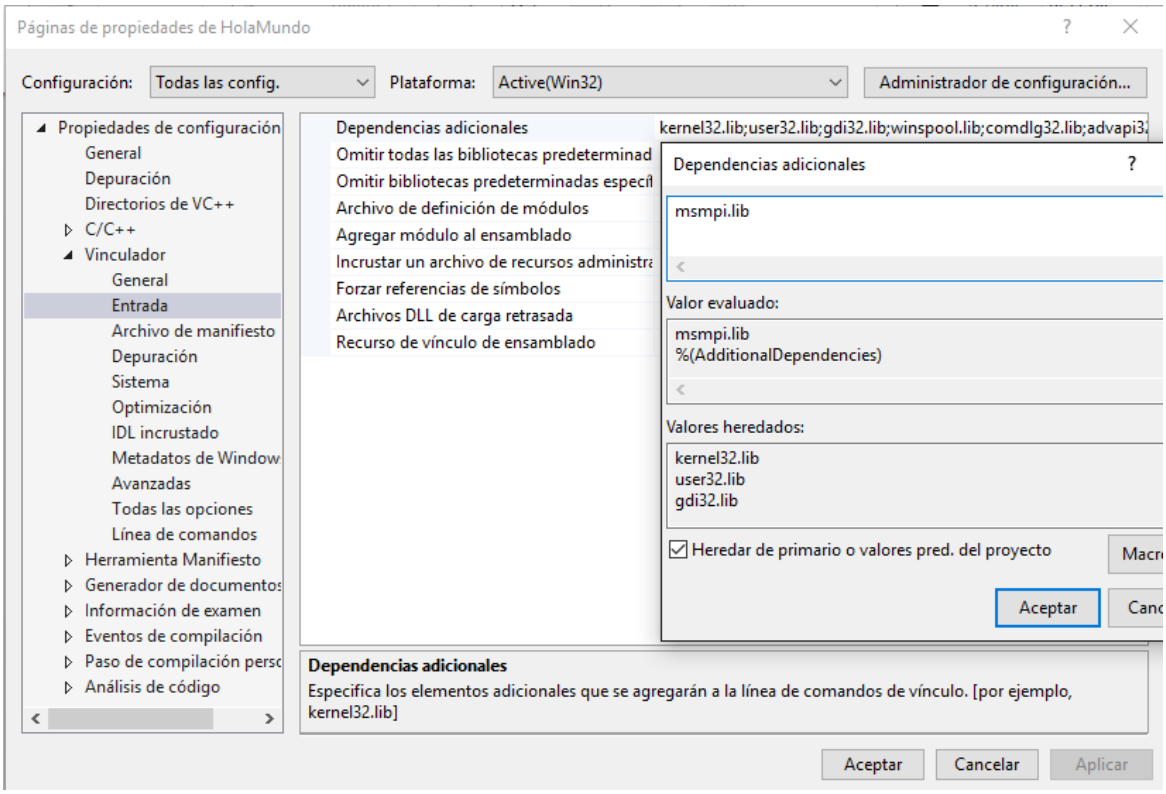
2. Configurar el Nuevo directorio include adicional que, si hemos aceptado la ubicación por defecto durante la instalación, se encontrará en: “C:\Program Files %28x86%29\Microsoft SDKs\MPI\Include” es importante notar que **no** se ha de seleccionar subcarpeta: x64 ó x86.



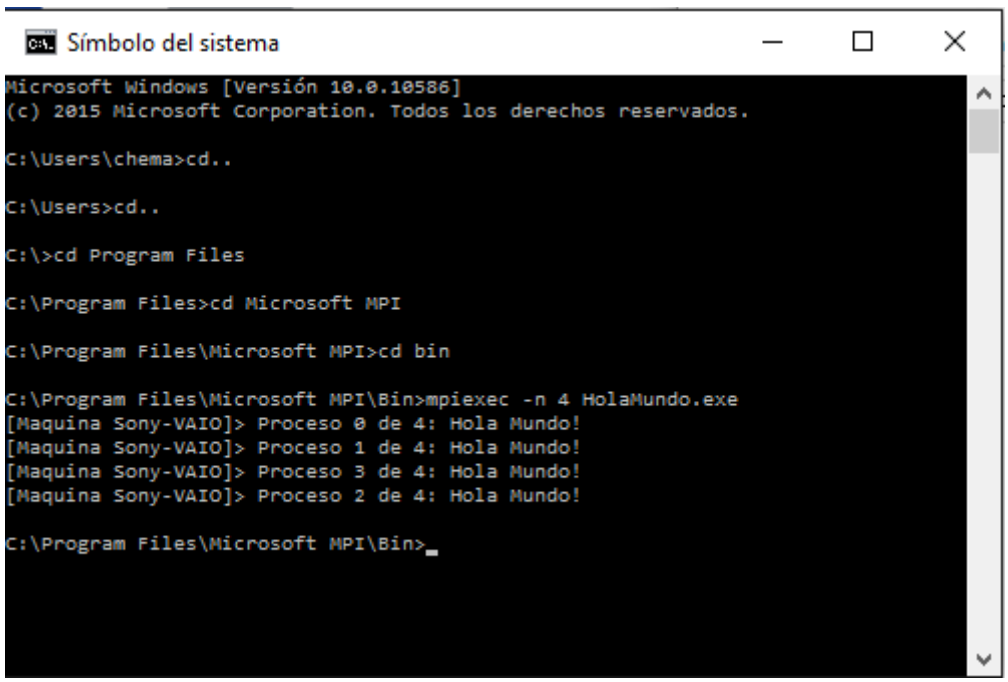
3. De forma análoga configurar la nueva carpeta lib. Ahora **sí** es necesario seleccionar la subcarpeta x86 para aplicaciones de 32 bit o x64 para las de 64 bits. Por defecto trabajamos en 32 bits, pero será interesante comprobar el efecto que tiene la compilación en 64 bits en el rendimiento del sistema.



4. Ajustar también la nueva librería MPI.



5. Ahora ya se puede construir la nueva solución como siempre. Para ejecutar el programa, el fichero .exe y el lanzador de MPI deben estar en la misma carpeta o si no se debe ajustar el path para que apunte a la ruta del lanzador. El lanzador es mpiexec.exe y se encuentra en Archivos de Programa > Microsoft MPI > bin. Escribe mpiexec –n np program.exe, donde np es el número de procesos que se pretende lanzar. El código correspondiente a este ejemplo lo podemos encontrar en el manual de Arquitecturas Paralelas.



Apécdice B: Envío de trabajos al cluster

Los trabajos que vamos a lanzar al cluster no se van a diferenciar de otras aplicaciones MPI que se hayan lanzado anteriormente en otros entornos. Trabajaremos en Windows 8.1 empleando roles de usuario que previamente han sido dados de alta en el dominio ARAVAN y en el cluster de computación, de manera que se encuentran autorizados para lanzar trabajos. Asimismo vamos a utilizar la implementación MPI de Microsoft: MS-MPI.

La herramienta que nos permite lanzar estos trabajos se denomina “Job Manager” y forma parte de las utilidades de cliente que incluye el HPC PACK 2012 R2 que se ha instalado en las máquinas. Antes de ver cómo se envía un trabajo al cluster, es necesario tener en cuenta una serie de consideraciones que afectan a nuestros programas:

1. No vamos a disponer de interfaz gráfica. La información de inicialización se va a pasar a las aplicaciones en línea de comando; la información en tiempo de ejecución se ha de leer de fichero. Será necesario modificar el programa en cuiertos casos para que acepte y lea la información en línea de comando. En el caso de la multiplicación de matrices, el tamaño de éstas se va a suministrar de esta manera. Mediante una sentencia como: “size = atoi (argv[1]);” podemos introducir el tamaño de la matriz en línea de comando y recuperarlo para operar dentro del programa.

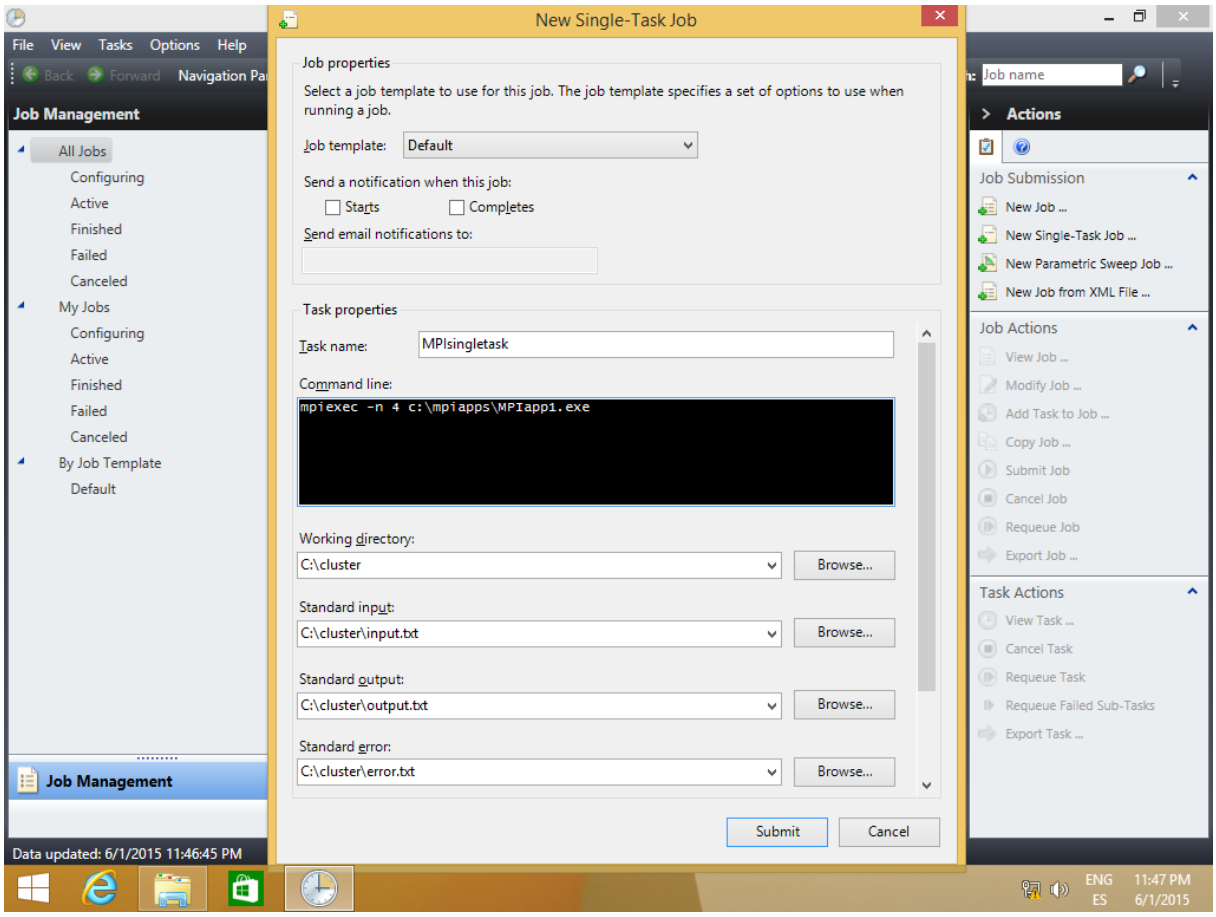
2. La salida estándar se va a redirigir a fichero, que tendremos que abrir y analizar una vez haya finalizado la aplicación.

3. Para que el gestor de trabajos no nos genere un error en su realización, deberemos finalizarlos con código de salida cero. Esto se logra mediante la función “exit (0)” que se encuentra definida en <stdlib.h>.

4. En MS Visual Studio vamos a generar las aplicaciones bajo la configuración “Release”

Generación de trabajos en Job Manger para la máquina local.

Genéricamente un trabajo está formado por una serie de tareas, siendo éstas las aplicaciones que el usuario quiere ejecutar. En nuestro caso vamos a lanzar trabajos con una sola tarea: nuestra aplicación MPI. Para este caso particular existe la opción de crear trabajos con una tarea (“Single Task Job”).



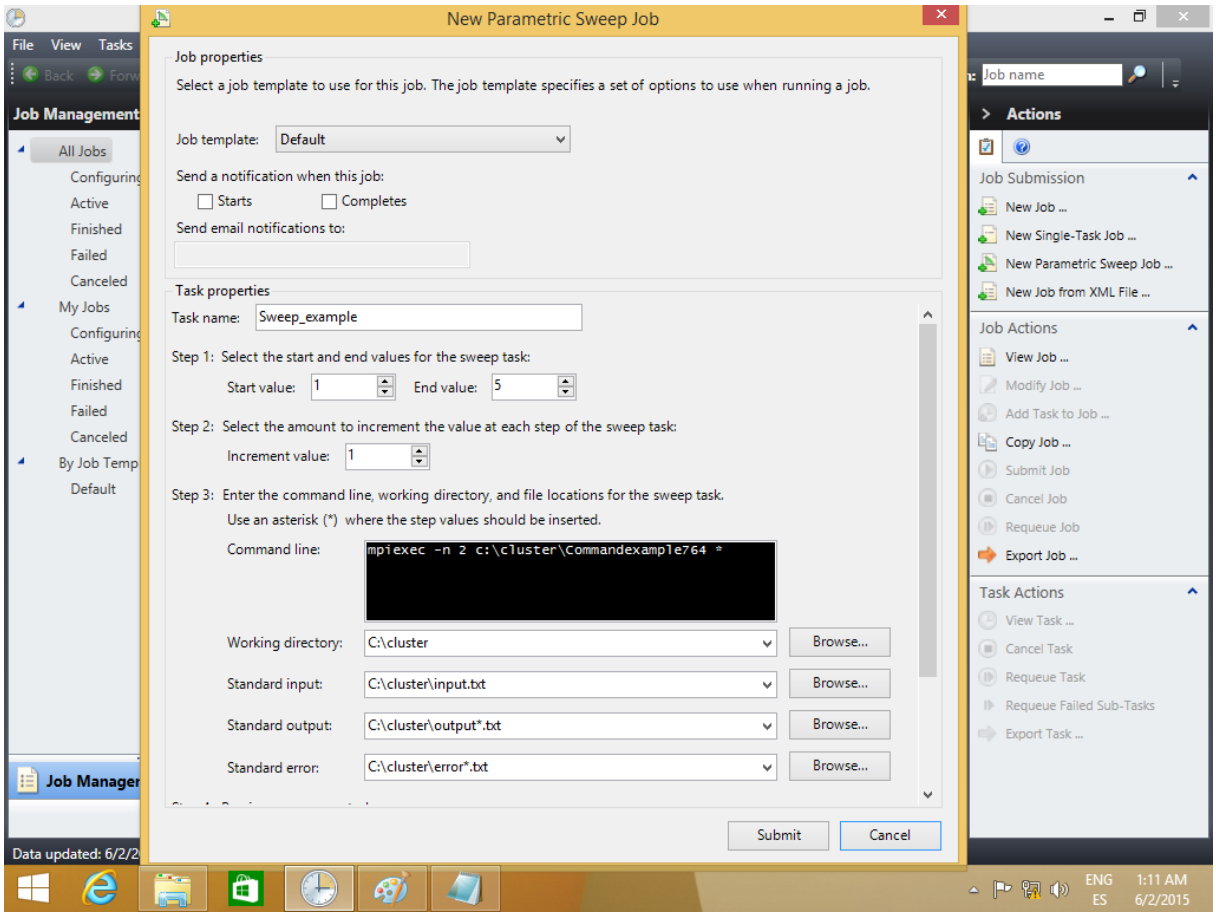
Configuración de trabajos con una sola tarea.

Vemos que se ha de configurar el directorio de trabajo por defecto que en nuestro caso coincide con la carpeta en la que vamos a ubicar los ficheros de texto asociados a la tarea. Configuramos cada uno de estos ficheros. Si no se van a emplear datos de entrada, no es necesario configurar la redirección de la entrada estándar.

En la línea de comando se define la tarea a llevar a cabo, que en este caso es una aplicación paralela MPI: "mpiexec –n 4 c:\mpiapps\MPIapp1.exe". La aplicación no tiene porqué encontrarse en el directorio de trabajo. El parámetro "-n 4" indica que queremos lanzar 4 procesos.

Trabajos de barrido paramétrico.

En muchos ámbitos de trabajo las tareas no se realizan de forma aislada, sino que se llevan a cabo tareas relacionadas de las que se extraen resultados que se analizan de forma comparativa, complementaria, etc. Por ejemplo, nuestra multiplicación de matrices la ejecutamos sobre diferente número de nodos y con matrices de tamaños diferentes. Es posible lanzar un trabajo para cada caso, obviamente, pero es más eficiente configurar una batería de pruebas y lanzarla de una sola vez. Esto es lo que nos permite hacer la opción de gereración de trabajos de barrido paramétrico ("Parametric sweep job").



Configuración de trabajos de barrido paramétrico.

En el ejemplo hemos configurado que el parámetro va a variar de 1 a 5 en incrementos unitarios. Se realizarán, por lo tanto, 5 tareas, una para cada valor del parámetro. Es interesante observar que el asterisco que ubica el parámtro en la línea de comando, también se ha añadido a los nombres de los ficheros de salida. Esto permite que cada tarea imprima en su propio fichero sin sobreescribir la salida de otras.

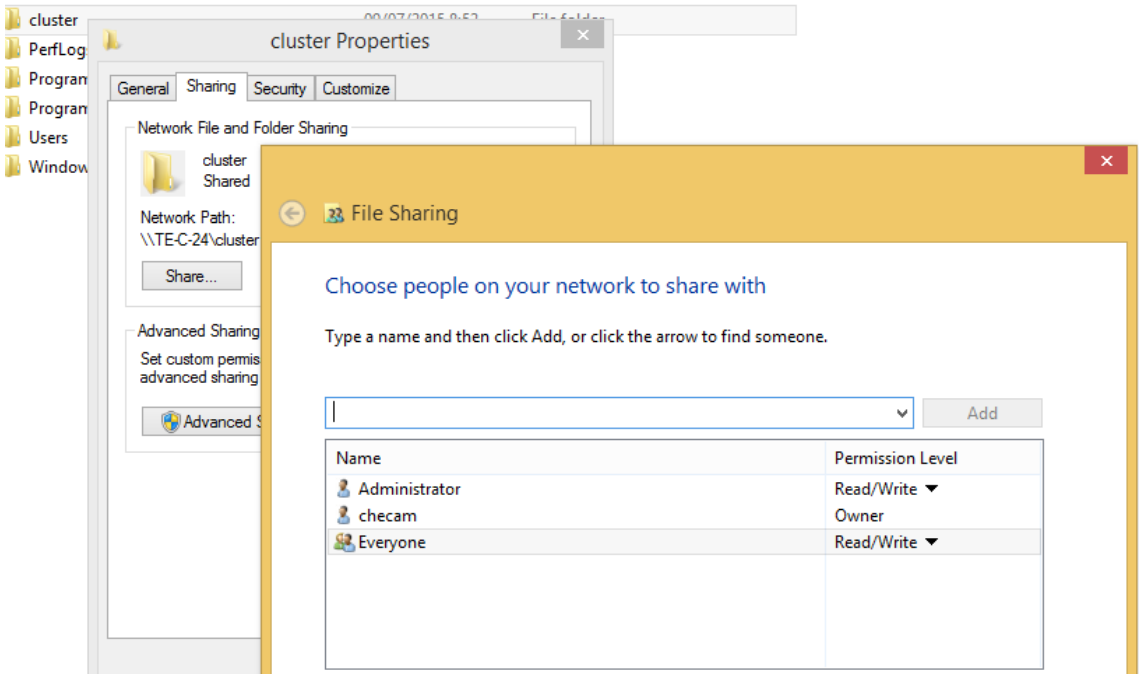
En este caso hemos asociado el argumento introducido en línea de comando al parámetro de variación pero en realidad se puede asociar a cualquier aspecto. Por ejemplo, si quisiéramos modificar el número de procesos que ejecutan la tarea en cada caso, como hacemos en las pruebas de rendimiento sobre la multiplicación de matrices, podríamos escribir la siguiente línea de comando: “mpiexec –n * c:\ruta\multimatriz 5000”. También es posible emplear más de un asterisco pero no suele ser interesante porque no suele suceder que sean necesarios los mismos valores en dos posiciones diferentes de la línea de comando. Lo que no es posible es anidar barridos. En el caso que nos ocupa, podemos variar el número de procesos o el tamaño de las matrices pero no ambas cosas de manera anidada.

Generación de trabajos en Job Manger para el cluster.

Conceptualmente no existe diferencia entre lanzar un trabajo en la máquina local y en el cluster, ya que una es parte del otro. Sin embargo hay algunas cuestiones de configuración que merece la pena destacar en este aprtado:

- Compartición de carpetas.
- Configuración de rutas.
- Selección de nodos.

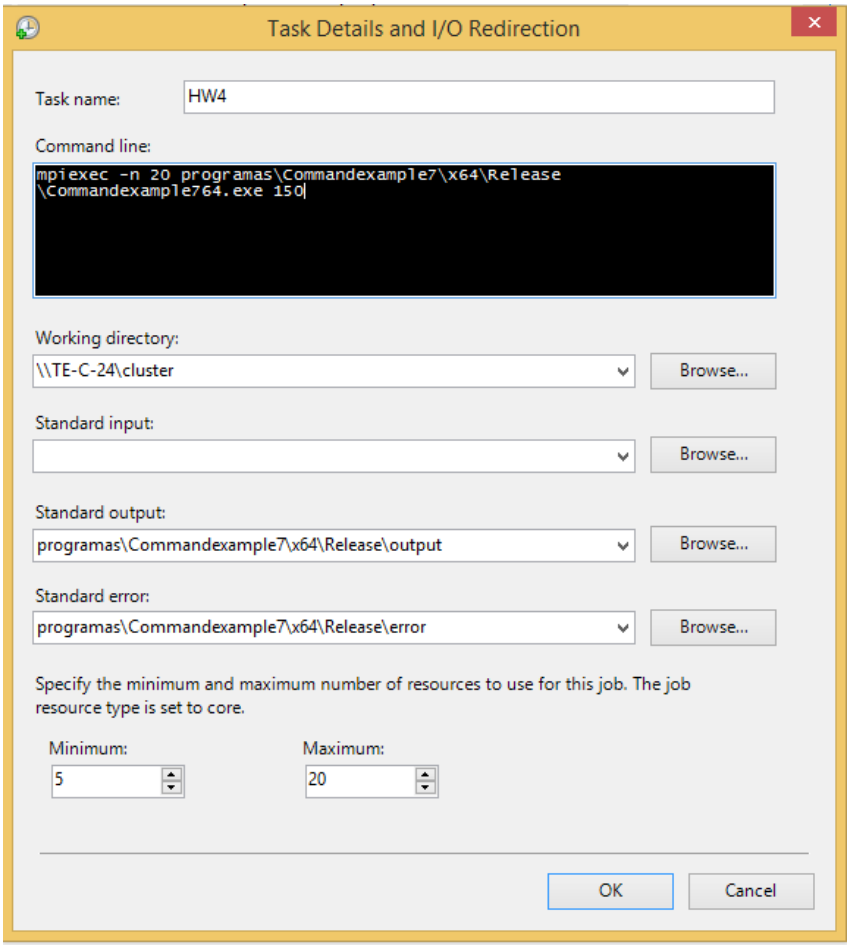
Es necesario que la carpeta y subcarpetas en las que se va a trabajar seencuentren compartidas. La compartición la podemos realizar mediante el Explorador de Windows, editando las propiedades de la carpeta correspondiente al directorio de trabajo.



Compartición del directorio de trabajo.

Los usuarios que tengan que poder ejecutar el trabajo deberán disponer de los permisos correspondientes.

Si el trabajo ha de ser ejecutado por otros nodos, la ruta debe ser conocida por todos ellos bajo una denominación común. Para ello se utiliza el formato UNC (https://msdn.microsoft.com/en-us/library/gg465305.aspx). Bajo este formato se configura la ruta del directorio de trabajo; el resto de rutas: ficheros de entrada y salida y aplicación a ejecutar en la tarea, se consideran referidos a dicha ruta. La figura 6.4 muestra cómo llevar a cabo esta configuración.



Configuración del directorio de trabajo compartido.

En este caso, la ruta completa de la aplicación sería:

*TE-C-24**c:**cluster**programas**Commandexample7**x64**Release**Commandexample764.exe*, siendo "150" un argumento en línea de comando de la aplicación.

Finalmente, se ha de realizar la selección de los nodos en los que se pretende correr la aplicación. Obviamente, se trata de un máximo, ya que si la aplicación no crea suficientes procesos/hilos de ejecución para llenarlos, no se van a utilizar algunos.

En la configuración del trabajo, acudiremos a la opción "Resource Selection" para definir los nodos que queremos habilitar.

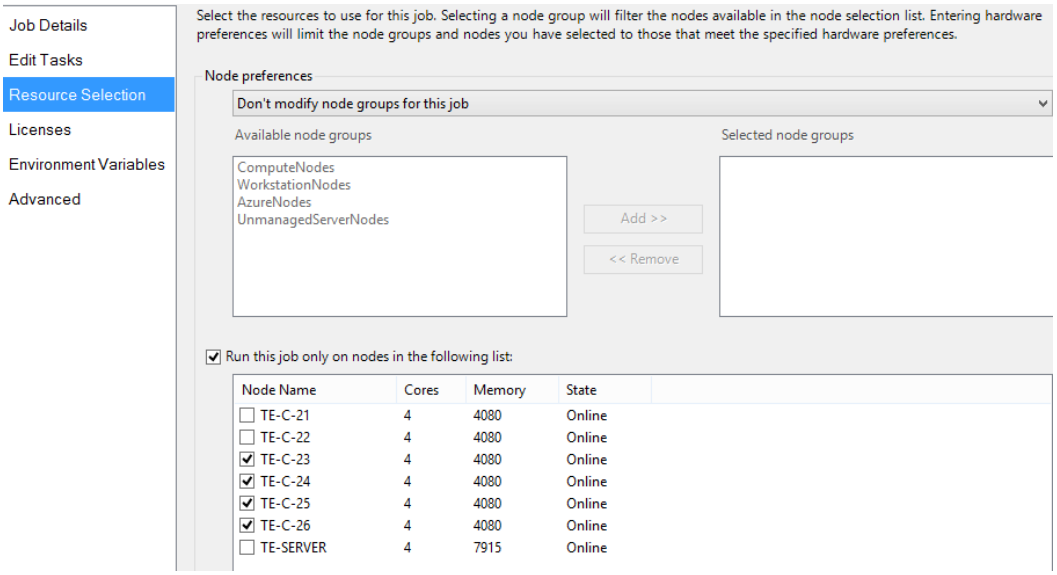


Figura 6.5. Selección de nodos.